Inhalt

[1. General notes: 4](#_Toc95363882)

[1.1 Inline functions: 4](#_Toc95363883)

[2.1 Header files: 4](#_Toc95363884)

[3.1 4](#_Toc95363885)

[2. Memory: 4](#_Toc95363886)

[4.1 Cache allocator 5](#_Toc95363887)

[5.1 Page table: 5](#_Toc95363888)

[3. Kernel: 5](#_Toc95363889)

[4. Kernel Mode VS User Mode: 5](#_Toc95363890)

[5. Process: 5](#_Toc95363891)

[6.1 Process termination: 6](#_Toc95363892)

[7.1 Process state: 6](#_Toc95363893)

[8.1 task\_struct: 6](#_Toc95363894)

[9.1 Init process: 6](#_Toc95363895)

[10.1 Daemon process: 6](#_Toc95363896)

[6. Scheduler: 6](#_Toc95363897)

[7. System call: 8](#_Toc95363898)

[8. I/O on file: 8](#_Toc95363899)

[11.1 Open system call: 8](#_Toc95363900)

[9. File descriptor: 8](#_Toc95363901)

[10. Process: 8](#_Toc95363902)

[11. Threads: 9](#_Toc95363903)

[12.1 Mutex: 9](#_Toc95363904)

[13.1 condition variables: 9](#_Toc95363905)

[14.1 Thread Cancellation: 9](#_Toc95363906)

[12. Data structure: 9](#_Toc95363907)

[15.1 Linked list: 9](#_Toc95363908)

[16.1 Koject: 10](#_Toc95363909)

[13. Interrupts: 10](#_Toc95363910)

[17.1 Data structure in Interrupt module: 10](#_Toc95363911)

[18.1 Bottom half: 10](#_Toc95363912)

[14. Shared library: 11](#_Toc95363913)

[15. Time: 11](#_Toc95363914)

[19.1 RTC: 11](#_Toc95363915)

[20.1 Real Time: 11](#_Toc95363916)

[21.1 system timer: 11](#_Toc95363917)

[22.1 Timers 11](#_Toc95363918)

[23.1 Delay 11](#_Toc95363919)

[16. Signals: 12](#_Toc95363920)

[17. VFS: 12](#_Toc95363921)

[24.1 Proc: 12](#_Toc95363922)

[25.1 Sys: 13](#_Toc95363923)

[18. Virtual address space: 13](#_Toc95363924)

[19. IPC: 13](#_Toc95363925)

[26.1 Data transfer: 13](#_Toc95363926)

[20. System V: 13](#_Toc95363927)

[27.1 System V methods: 13](#_Toc95363928)

[28.1 system V keys: 14](#_Toc95363929)

[29.1 System V message Queues: 14](#_Toc95363930)

[30.1 System V Semaphores 14](#_Toc95363931)

[31.1 System V shared memory 14](#_Toc95363932)

[21. PIPES 14](#_Toc95363933)

[22. Message queues: 15](#_Toc95363934)

[32.1 shared memory 15](#_Toc95363935)

[33.1 Critical sections: 15](#_Toc95363936)

[34.1 Semaphors: 15](#_Toc95363937)

[35.1 Atomic operations: 15](#_Toc95363938)

[36.1 Spinlocks: 15](#_Toc95363939)

[23. Device Drivers: 15](#_Toc95363940)

[37.1 Resources management: 16](#_Toc95363941)

[38.1 Bus system: 16](#_Toc95363942)

[24. System calls: 16](#_Toc95363943)

Books to read:

09.02.22 Linux Device Drivers Development Develop customized drivers for embedded Linux by John Madieu

Linux Driver Development for Embedded Processors - Second Edition Learn to develop Linux embedded drivers with kernel 4.9 LTS by Alberto Liberal De Los Ríos (z-lib.org)

Systems Programming in UnixLinux by K. C. Wang (z-lib.org)

15.02.22 Essential Linux Device Drivers by Sreekrishnan Venkateswaran (z-lib.org)

20.02.2022 Linux Kernel Development, 3rd Edition A thorough guide to the design and implementation of the Linux kernel by Robert Love (z-lib.org)

21.02.2022 Professional Linux kernel architecture by Wolfgang Mauerer (z-lib.org)

21.02.2022 Linux-Kernel-Programming-Kaiwan-N-Billimoria

23.02.2022 Mastering Embedded Linux Programming Create fast and reliable embedded solutions with Linux 5.4 and the Yocto Project 3.1 (Dunfell) by Frank Vasquez, Chris Simmonds (z-lib.org).pdf

23.02.2022 Embedded Linux system design and development by P. Raghavan, Amol Lad, Sriram Neelakandan (z-lib.org)

24.02.20211 Understanding the Linux Kernel, Third Edition by Daniel P. Bovet, Marco Cesati (z-lib.org)

1. General notes:

UMA: unified memory access -> all processors can access all memory

NUMA: each processor has a memory which he can access fast and can access another memory which is local for the other processor but slower

Void \* = unsigned long -> as void is architecture dependent Linux uses long to make sure that the code will be the same regardless of the architecture

Kernel code is not compiled against c library, it has header file which is inside include folder in linux source file

1. Inline functions:

Normally it is written inside header file and then included in the file which is going to use it as u must define the function before u can use it

1. Header files:

* It is inside include directory in Linux source tree

1. Kernel configuration and make file:

It would be stored in the root of the source code at .config file

Default configuration would be under arch/<arm>/configs/\_\_defconfigs

To copy the defconfig in .config

* + make ARCH=arm \_\_defconfig
  + Make menuconfig

To build the kernel:

* + Make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- zimage

To build the module:

* + Make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- modules

To install modules:

* + Make arch=arm CROSS\_COMPILE=arm-linux-gnueabihf modules\_install
  + This command expects environmental variables INSTALL\_MOD\_PATH or it would install it under /lib/modules/$(KERNELREKEASE)/kernel

1. User space and kernel space:

Processor is can in one of three places at any time

* + User space running user code
  + Kernel space in process context running on behalf of processor executing system call
  + Kernel space in interrupt context executing interrupt

1. Kernel space:

* Memory space owned by the kernel and protected by access flags
* Kernel can access both kernel space and user space

1. User space:

* Where the normal programs run, in user mode CPU access only memory owned by the processor

1. Monolithic:

* Kernel is a single process with one address space and the communication with the user land happens via system call

1. Modules:

* To support modules kernel must be build with the following option enabled
  + CONFID\_MODUKES=y

1. Helper functions:
2. container\_of:
3. Memory:

Kernel divide physical memory to Nodes -> zones -> pages

Zones are -> DMA, normal and high memory

Pages is the virtual memory unit while frame is the physical memory unit

Struct page represents the physical memory

Pfn: page fram number which refers to the number which is given to each frame in the memory

The Virtual Memory is divided 3:1 -> lower 3 GiB is for user mode while the upper 1 Gib is for Kernel mode

Fourth GiB address space is for the kernel starts from, \_\_page\_OFFSET

There is two micros to convert between physical and virtual address in kernel land

* + \_\_pa() -> would return physical address
  + \_\_va()-> would return virtual address

Allocation of memory is done using alloc\_pages() -> in returns physical address

To convert the physical to logical address page\_address()

Get\_free\_pages() -> will return logical address()

Kmalloc() -> allocate memory which are continuous physically and virtually in the kernel space not in a process space

Vmalloc() -> allocate memory which is continuous virtually

Task\_struct represent a process which has mm\_struct so current->mm points to the memory mapping table

Mmap system call: u makes this address maps to this area in ram

1. Slab allocator:

* The one which kmalloc() uses
* Slab: it is array of object in the memory with the same size which can be full, empty or partial full
* Cache: it is a linked list of slabs

1. Cache allocator

* Cache stores different type of object
* Slab is one or more page
* Slabs contain object
* Cache is full, empty, or partially and contains slab

1. Page table:

* Every process has it’s own page table, and it is managed by the operating system
* It is used to create a relationship between virtual address space and the physical address space
* Addresses in virtual memory is divided in 5 parts, 4 for the table and 1 for the offset
* BITS\_PER\_LONG -> define how many bits it is used to define long data type

1. MMIO memory mapped I/O:

* Kernel used HIGH\_MEM to map I/O the device register
* U need to map the address to the kernel address space using ioremap
* \_\_ iomem protect us from using the wrong driver to access the memory which is I/O memory

1. VFS:

It enables different file system to exist and cooperate with each other

It represents files and data

It allows different system call to work regardless of which file system is available

Virtual file system makes linux support different physical file system without a problem for example user space write would call sys\_write”which is VFS” which would call write in the physical media

Linux organize file system using 4 concepts

* + File: which is collection of bytes
  + Folder: which is file which has information about the files
  + Inode: it has the information about the file such as size, permission, etc, it is stored as superblock which has information about the file system as a whole

1. Proc:

* It is on the fly file system
* It is used to communicate with the kernel
* It is used for:
  + Memory management
  + File system
  + Device drivers
  + System bus
* The main purpose of Proc is to deliver process data
* Every entry is an instance of proc\_dir\_entry

1. Sys:

* It is on the fly file system

1. Virtual address space:

Virtual address space of a process has sections:

* + Text segment where there is a code
  + Stack
  + Heap
  + Data

Address space of the memory is done in the elf file

Memory mapping:

1. Kernel:

Resource Manager

Scheduling

Memory management

Provide system API

Provide file system

It has no access to C library or standard header file

<linux/notify.h> -> would be in include/linux/notify.h

1. Kernel Mode VS User Mode:

Kernel mode can access the data in user and in kernel land

User mode can access the data only in user land

1. Linux device model kobject:

To provide features like class and reference

It made communication with user space is unified via sysfs

Bus:

* + It is responsible for matching mechanism which will be called every time a device or driver is registered to the bus
  + Probe: it is called when the match has happened

Kobject:

* + The core of device model.
  + It brought OO programming to the kernel
  + It enables reference counting of the objects
  + Sysfs representation
  + It is abstact class
  + It has name which will represent the name which will be shown in sysfs
  + Get/put increment /decrement the reference count of kobject
  + Every kobject has kobj\_type which provide release/sysfs\_ops and attribute, it could be provided by kset
  + Parent would point to the parent in sysfs system and it is the kset

Kset:

* + It is used to group kobject in a set
  + It is always represented in sysfs
  + Each keset has subsystem ex: block, device

1. Process:

Instance of executing program

Kernels identify the process by PID

Kernel schedule thread not the process

Kernel loads it in the memory and provide resources for it

Process can create another program by calling fork()

Kernel will duplicate text, data, stack, heap of the parent to the child

The child calls execve() to replace the old data, stack heap, text

1. Process termination:

* Process can be terminated by calling exit or be receiving killing signal
* The parent can get the exit status of the process by calling wait(), the child process will end when it calls exit() or when it reaches it is end
* Fork() -> will return 0 in the child case and -1 on error, and child ID in parent case

1. Process state:

* It could be running, waiting, sleeping, zombie, over

1. task\_struct:

* this is how the process is represented in the kernel
* it has all the information the kernel needs to work with the process
* it is double linked list
* it represents a thread
* it has thread\_info structure
* it has the location of the stack
* it has element children which is a list head to the childrent structure which u can use to go through the whole process

1. Init process:

* it has the ID 1
* it cannot be killed

1. Daemon process:

* it runs in background
* long live process

1. Scheduler:

Scheduler can be cooperative or preemptive:

* + Comparative the processer itself decide that it should stop and yield itself to the scheduler
  + Preemptive: the scheduler decides that it the processor has exceeded it is time slice and stop the processor from being running

Every element which is needed to be scheduled must have schedule entity which have elements which enable scheduler to use it to perform scheduler

* + Scheduler\_entity: load, node which is used to represent the entity in red black tree, execute time which save the current the entity has been active

Process can be I/O or processor bound

The idea is that the high priority should run first and longer than the process with less priority

Priority is nice from -20 to 19 and real priority from 0 to 99

Real time priority which is number from 0 to 99 the higher the number the more priority it has

CFS: complete fair scheduler

Time slice: it is the amount of the time the process is going to give to run

CFS: gives a slice of the time slice to each process according to nice value

Load of the system is going to be a function of nice value of all runnable processes the scheduler is going to give every process a portion of the time slice according to it’s nice value

The problem with scheduler:

* + If u map the priority to the time slice, then u won’t get a good result as the time different between two process which their value are 0 and 1 won’t be the same as the different between two processors which have the value 18 and 19 t
  + the solution is -> u decide default time slice which is for example 20 mile second which means u have to run all the processes in 20 mile seconds and each process would run for a part of the time slice which is an equation of the whole time slice and the nice value of this process and the nice value of all the others processes

Preemptive concept:

* Process can preemptive another process in user land
* If the kernel is in system call no process can preemptive it rather than an interrupt

1. Scheduler classes:

* Normal class -> time slice is function of priority that means even less priority process will get to be active for some time
* Real time -> the higher priority class only gets to be active, and it leaves only when it yields the process “sleeps”

1. Calling scheduler:

* When the processor sleep or the system time calls the scheduler
* The process has three type of priority: prio,static,normal:
  + Prio: it is what the scheduler uses
  + Static: what was given when the process was created
  + Normal: it is function of static and scheduler

1. System call:

It an entry point from user land to kernel land

Application uses c library functions to execute system call

The c library will use int machine instructions to makes the CPU jump to a specific place in kernel land which will have routine to handle system call and will update s register in cpu which has system call number, if error happens y library will update errno to say that error has happened

If the system call failed it will return -1 and update errno variable, u should check only the errno if the system call returned -1 as a successful system call won’t reset errno

1. I/O on file:

File descriptor referres to all opened files (PIPE, FIFO, SOCKET)

Any process has 3 standard file descriptors

* + 1 -> stdin
  + 2 -> stdout
  + 3 -> stderror

To change the standard file descriptor use fropen()

1. Open system call:

* Flag will be access mode and to create the file or truncate it
* Mode will be the permission in this file which makes sense only if u create the file for the first time

1. File descriptor:

There is not one to one match between file descriptor and open file

Using Open, dup, or fork will create new file descriptor which will refer to the same global file descriptor which will refer to the same i\_node

An example of dup is redirect of error message 2>&1 -> dup2(2,1)

1. Process:

Every process has process id which u can get by calling getpid(void)

To get the parent of the process u can call getppid(void)

Kernel divide physical RAM into frame ad Virtual Ram into Pages

Page -> VM / Frame -> Physical

There is a page table between page and frame for each process which is maintained by the kernel

Process is created by using fork() which create a process as a copy of the parent process

Execv() will replace the process with a new process

U can assign handler to be called on exit()

System() -> allow writing shell in C script

1. Threads:

Every thread has errno variable to prevent race condition

Pthread\_t -> datatype for a thread

Pthread\_create() -> will create a thread and execute it

Pthread\_exit() -> can be called by any functions which is called from pthread\_create

Pthread\_join() -> wait on a specific thread to be joined cancelled

Pthread\_detach() -> to make the thread not joinable to save resources in the system

If the main exit() all the process are cancelled

1. Mutex:

* It can be statically or dynamically created
* static pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER;
* s = pthread\_mutex\_lock(&mtx)
* or pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr);
* int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);
* attribute:
  + PTHREAD\_MUTEX\_RECURSIVE -> count. When a thread first acquires the mutex, the lock count is set to 1. Each subsequent lock operation by the same thread increments the lock count
  + s = pthread\_mutexattr\_settype(&mtxAttr, PTHREAD\_MUTEX\_ERRORCHECK);

1. condition variables:
2. Thread Cancellation:

* Pthread\_cancel(pthread\_t thread) -> to cancel a thread
* Pthread\_cancelstate() -> u can make the thread not cancellable, or u can assign the thread that it only cancel when it reaches a specific point

1. Data structure:
2. Linked list:

* The first element is called head
* The list\_head structure is embedded in every element in the list with the name list
* List\_head it has prev and next to point to the next and the last element we need to point to
* List\_entry() -> use container of to get the struct which has the embedded element
* To initialize the element in the list we use 2 possible ways:
* INIT\_LIST\_HEAD() -> dynamic
* LIST\_HEAD\_INIT() -> static
* To create the head : LIST\_HEAD(name\_of \_head)
* List\_add() -> add after the head
* List\_del() -> to delete element from the list
* To iterate over the whole list and get the struct use list\_for\_each\_entry(whole struct, struct head, name\_of\_head)
* List\_for\_each\_entry\_safe() -> u can delete the element and keep iterating over the list

1. Sleeping mechanism:

* DEFINE\_WAIT() -> will create a wait queue entity which we can be added to wait\_queue

1. Koject:

* The role of kobject ist:
  + Providing reference count
  + Locking sets
  + Export properties using sysfs
  + Managing lists of objects
  + Kobject is embedded in the structure

1. Interrupts:

Interrupt enables the hardware to signal the operation system that something has happened

Interrupt controller: device which multiplex interrupt lines to a single line in the processor

IRQ: interrupt request which is an integer number to distinguish between different sources of interrupt

ISR: interrupt service routine which will run when the interrupt is fired

Interrupt handler runs in interrupt context without process which he is related to so it cannot sleep

Interrupt handle while other interrupts are disabled that is why it has to be really fast to avoid the situation where the interrupt might be lost

To register a handler to a specific interrupt request\_irq()

* + If u used shared interrupt u have to path a unique value to dev so the kernel knows which one to remove if u decided to free the interrupt line

1. Data structure in Interrupt module:

* Kernel uses hw\_irq\_controller to abstract IRQ controller
* Irq\_chip: it is used to enable, disable, end
* Irq\_action is to represent handler

1. Bottom half:

* It runs while another interrupt as enabled
* Softirq:
  + It is an array of pointers of functions which will be called later

1. Tasklet

* Only One instance of the same tasklet can run simultaneously
* Declare tasklet using tasklet\_init()
* Enable tasklet using tasklet\_enable()
* Schedule using tasklet\_schedule()

1. Work\_queue:

* It is the only choice when u want to sleep on the bottom half
* There are 2 options, to use the shared work queue or use a dedicated thread for this work\_queue

1. Shared library:

Place a function in single place which can be used by multiple process at the same time

1. Time:

Tick: smallest time unit in the system

HZ: how many ticks in one second

HZ affects the timer interrupt which affects the scheduler

Jiffies: number of tickles since booting

1. RTC:

* Real time clock
* Nonvolatile device which saves system time
* It keeps tracking of the time even if the power is off
* On boot the kernel reads RTC and use it to update wall time

1. Real Time:

* It is calendar time or wall clock time (time elapsed since a specific point)
* gettimeofday(timeval) -> assign the seconds and millisecond since EPOC in time\_t
* time(NULL) -> will return seconds since EPOC
* ctime(time\_t) -> return time as string wed Jun 8 14:22
* struct tm \* localtime(time\_t) -> get the time to broken down time year, month
* mktime() -> convert broken down time to EPOC time
* asctime() -> convert to wed Jun 8 14:22

1. system timer:

* it is used to provide a mechanism for driving the interrupt at periodic time
* PIT: programmable interrupt time
* Kernel program PIT to interrupt the processor at HZ frequency
* Every interrupt the kernel will update the wall time and fire the expired timers

1. Timers

* It is used to delay execution of some function until a later time in the future
* Struct timer\_list represent timer
* Init\_timer(&) -> to initialize the timer
* Assign the values after initialization the timer
* Add\_timer(&timer) -> to fire the timer so the timer starts counting
* del\_timer() -> the job is to prevent the timer to be fired in the future

1. Delay

* It is way to give the hardware time
* While (time\_before(jiffies, jiffies + 5));
* Udelay() -> it is used to delay for time without using jiffies as the processor is faster than jiffies the function is implemented depending on the time the processor is using to execute an instruction
* U can schedule the task to sleep using:
  + Set\_current\_state() , schedule\_timeout()

1. Signals:

Software interrupt

Process can deal with the signal in three different ways: Ignore, default or handler

Famous signals are:

* + SIGTERM: it is used to stop the process
  + SIGKILL: kill the signal u cannot assign handler to it so it makes sure that u end the process it is equal to KILL -9
  + U can assign a handler to a signal using signal() functions
  + U can send signal to another process using KILL(pid, signal) command

1. IPC:

It has three forms: signals, data transfer, synchronization

Communication facility:

1. Data transfer:

* process write, and another process read, it requires that the data go to the kernel as process write to the kernel while the other process reads from the kernel
* byte stream:
  + pipes, FIFO, datagram socket where each read may read arbitrary numbers of byte
* Message:
  + Message queue, datagram message where u can only read the whole message

1. System V:

It is connectionless which mean It has no handle (file descriptor)

Use integer key value not path name to identify the communication

Every system V mechanism has get() system call which is similar to open system call, it takes integer number which is analogous to file name

1. System V methods:

* get() which either create new IPC object with a given key and return identifier to this object or return identifier with this object
* ctl() which is used to preform operation on system V IPC such as delete IPC\_RMID, for message queue deletes happens immediate regardless whether any process is using it, while shared memory it is already deleted after all process detach it shmdt()
* kernel does not have reference to the object which means it does not know which process is using it

1. system V keys:

* it is represented using data type key\_t
* calling get on the key will create a IPC object
* u get the key by IPC\_PRIVATE which will create new IPC object

1. System V message Queues:

* Communication via message queues us message oriented, receive get the whole message
* Message has type which give the read the ability to read a message by type
* Msget(key, flag) -> create or get object identifier for a message
* Msgsnd / msgrcv perform I/O on a message queue with a specific msgid, the message has type to make it possible for the program to send a specific message to a specific process
* ,msgrvc -> if the type is 0 and the first message in the queue, positive number then a message of the specific type will be read
* Normally if there is no message the process would sleep unless u choice the flag IPC\_NOWAIT

1. System V Semaphores

* Operations on semaphores are adding, subtracting, setting to a value, waiting for the value to be 0
* To create semaphore semget() which will create a set of semaphore
* Semctl(int semid, int semnun, cmd, union semun):
  + This would specify the command which will be operated on the semaphore,
  + Semnun -> it is the semaphore index inside the index
  + Control are set,remove,getvalue,status
  + Semun.val is the value to set
* Semop(semid, sembuf,nsops)
  + Where sembuf is the operation to be performed sem\_op
  + -1 is to lock, +1 is to unlock, 0 is to check
  + Nsops is the number of sembuf

1. System V shared memory

* Kernel does not need to get involve a lot as the memory is going to be a part of the process memory
* Steps are:
  + Get to create the memory using identifier,
  + Shmat -> to make the memory part of the process address space
  + Shmdt -> to remove it from address space
  + Shmctl -> to delete the memory

1. PIPES

It happens when the kernel created 2 process and connect output of the first processor (file descriptor 1) to the input of the second processor (file descriptor 0)

It is byte streaming connection, there is no concept of message

It is one-way directions

Pipe() -> system call to create the pipe normally u then call fork to create second process

It is important to close the descriptor in pipe so the read process wont sleep on the pipe and will see EOF

1. Message queues:

Message boundaries are preserved

Each message includes integer type files to make it possible to select a message by type rather than in order

1. shared memory

* Shared memory: the same memory is seen from 2 processes
* It is faster
* There is a need for synchronization

1. Critical sections:

* Means only one process may enter this section at a given time

1. Semaphors:

* Number which can be positive or negative
* Two operations are available up and down
* The process call down if the variable is zero it can enter the section
* If the other processor called down it will sleep
* Down step is atom step
* Kernel puts the processor on sleep
* It works well in user land

1. Atomic operations:

* Atomic\_t -> it is the data structure to represent atomic variable

1. Spinlocks:

* It wont sleep if the lock is taken

1. Device Drivers:

I/O device:

* + Hardware must be addressed
  + User application must have tool to access the device
  + User space should know which device is available
  + I/O memory: port address of the I/O port is mapped into normal memory, kernel provided abstraction layer to map I/O to memory. CPU would go the I/O memory when u use this address

1. Char device:

* Each char device is represented by instance struct cdev
* Kernel keeps all the character devices in array which index is the major number and has function which can return cdev struct
* Cdev:
  + It represents char device
  + It has kobject, file\_operation, count

1. Platform device driver

* Platform driver handles device which does not set on conventional bus
* Everything must be done manually with platform drivers
* It is represented by struct platform\_driver which has probe, remove and driver
* To register a platform driver u have to call: platform\_driver\_register()
* Platform device is represented by struct platform\_device
  + Elements are: name, id, device, num\_resources , resources-> which is an array and num\_resource is the size of the array
* Platform\_getresources() can be used to get the resources from the device
* U can insert data in platform device in device.dev.platform\_data

1. Matching mechanism:
2. Resources management:

* All located io\_mem is children of iomem\_resource root

1. Bus system:

* Struct device is a way to represent device in kernel
* Struct device\_driver to represent a driver
* Struct bus\_type

1. Matching mechanism:

* It is the job of the bus system to perform matching if u register a specific device to the bus, it is the job of the bus matching to find the drivers which support this device and call it is probe function
* ID table matching:
  + it depends on device\_id structure which is different from bus to bus for example for i2c the structure is i2c\_device\_id
  + struct platform\_device\_id { char name[PLATFORM\_NAME\_SIZE];kernel\_ulong\_t driver\_data;

1. Device tree:

It is hardware description which describes with hardware as nodes with descriptions

Properties of the node could be empty or key value pair

Empty would describe Boolean properties

1. Properties:

* Each addressable device gets reg properties which describe the address range of the device
* #size-cells: how many 32bits is used to represent the size of the children node
* #address-cell: how many 32 bits is used to represent the address of the children node

1. Naming resources:

* As the drivers writes is not always the device tree write using index to represent the resources is not a good idea to avoid such case naming convention is introduced to make it easier to address the resources
* Reg-names: for reg properties
* Clock-names/interrupt-names

1. I2C:

It is master slave communication

1. Structure:

* I2C core:
  + Routine and data structure, which is available for STM, NXP to use it in their host adapter
  + It allows that the client maybe use another adapter as the interface is the same
* Host adapter:
  + It is bus driver which would be developed by STM, NXP they divide the code in adapter and algorithm
* Client driver:
  + By MUT to describe the client which should implement i2c-dev

1. API for I2C:

* I2c\_master\_send: S Addr Wr [A] Data [A] Data [A] ... [A] Data [A] P
* I2c\_master\_recv: S Addr Rd [A] [Data] NA S Addr Wr [A] Data [A] P
* I2c\_transfer: S Addr Rd [A] [Data] NA S Addr Wr [A] Data [A] P

1. Instantiate:

* Kernel code must instantiate i2c device as they are not enmurated in HW level
* Using I2C\_board\_info

The driver is represented as i2c\_driver and the clients are represented as i2c\_client

Probe functions:

* + initialize the data and register the device
  + it gets i2c\_device\_id which points to which device Id has been matched (this information is defined in the driver not in the device which has name, id property)
  + i2c\_device is a device which means it has all the functionality and property available for a device u can set data and get data using i2c\_get/set\_clientdata

1. I2C and device tree

* We need to define and register device and driver for i2C

1. GPIO:

SOC provide pin multiplexing which means a single pin has more than one function

Pin controller in Linux allows pin muxing(pinctrl) and pin configuration and enmeration

Pinmuxing : the pin us UART or SPI

Pin configuration: pull up or down, open drain

Most of the time every pad has up to 8 modes as the multiplex has 3 inputs

Pin control subsystem is nothing but a way to gather pins and pass then to pin controller driver

Pin controller driver apply the configuration to the pin usually it needs two nodes the first describe the function while the second says how to configure them

For software pin is nothing but line which can have only two value 0 or 1

U need to claim the I/O before using it

Linux enumerate all the pads using integer therefore bit notation must be mapped as integer

1. Linux pinctrl subsystem:

* API which is important to the SOC vendor to expose pin capability
* API for the driver to request the pin
* Interaction with the driver
* struct pinctrl\_pin\_desc {unsigned number; const char \*name;};

1. GPIO:

* Subsystem to control the GPIO
* GPIO is represented by integers or by descriptor
* GPIO descriptor:
  + Consumer of GPIO in dt should be name-gpio
* gpio\_get() -> get gpio for a specific function, it depends on function name and u can configure also the gpio as input or output it will return description an example of GPIO node : led-gpio = <&gpio1 15 GPIO\_ACTIVE\_HIGH>

where:

* led -> it is the function name
* gpio1 -> it is the GPIO controller in SOC level
* 15 -> it is the pin number in GPIO controller
* To get the gpio use
  + Gpio\_desc \*goip\_get
* To release gpio uses
  + Goip\_put()
* You acquire the GPIO to configure it

1. Steps:

* Stm write c file which define the name of the soc Pin names(integers) and the name of the I/O controllers compatible string which should be provided in device tree file

<https://code.woboq.org/linux/linux/drivers/pinctrl/freescale/pinctrl-imx7d.c.html>

* In the probe functions the assign [imx\_pinctrl\_desc](https://code.woboq.org/linux/linux/drivers/pinctrl/freescale/pinctrl-imx.c.html#174imx_pinctrl_desc) with the details which were available in the c file of STM

1. Embedded linux:
2. Component:

* Tool chain: to create software for the target
* Bootloader: initialize the board
* Kernel: manage resources
* Root file system: library that run on the kernel

1. Tool chain:
2. Components of the tool chain:

* Binutils: binary utility including assemble
* Compiler:
* C library: it has API for the operating system
* Header file: it has the definition for the micros

1. C Library:

* Libc: it has open, close, print
* Libm: for math
* Librt: shared memory

1. Bootlooder:
2. The role of bootloader:

* Initialize the system as in the beginning only CPU, static memory and ROM code are available
* Load the kernel in the ram and pass the hardware description to the kernel
* Pass command line to the kernel so it can change it is behavior

1. Sequence of bootloader:

* First stage:
  + The soc would come with ROM code which is not open source would try to copy code from one of the sources to the static memory:
  + Then the SPL would be ready to be used ROM code jump to it
  + At the begging the DRAM is not ready to be used
  + The place where the EOM code might try to search for SPL are
    - First page of flash
    - Partition in eMMC
    - UART
* Second stage:
  + Initialize the DRAM and copy UBOOT to it and jump to UBOOT
* Uboot:
  + Copy the kernel and device tree, and provide command line to the kernel
  + It can be used to update the kernel in the flash

1. The kernel:
2. Job of the kernel:

* Manage resources, deal with hardware, provide interface to user space
* User space communicate with the kernel via c library which has system call which is nothing but software interrupt

1. RootFS
2. Components:

* Init
* Shell
* Daemons
* Shared library
* Etc
* Modules in /lib/Modules

1. Storage system

It can be NOR or NAND or eMMC

1. NAND:

* Depending on how many bits are stored in one cell SLC-> single level cell means 1 bit per unit cell which is the best
* It is divided to erase block which can be 16 to 512 Kibs
* Write and read is divided in pages which is 2 or 4 kibs
* MTD divided the flash to 256 or 512 byte page to read and write
* There is OOC which 64 Kibs per page
* ECC is the way we use to know bad block

1. FTL:

* Flash translation layer provide:
  + - Sub allocation smaller allovcaion than the one which is provided by the hardware
    - Error correction routine
    - Bad block handling marks good block as bad if u cannot erase

1. MTD subsystem:

* Core which provides API to the BSP
* BSP which is the low level for the flash
* User ex: is JFFS2

1. ECC

Error correction code

1. Reptation:

* Save that data three times and make majority vote and to decide which data is right
* If u repeat the message three times means u can fix only the error if it there is 2 errors by one bit

1. Hamming:

* Every 4 bits would have 3 bits which is xor of the 3 bits u can find where the error is only if one bit was flipped
* U can correct 1 error in 4 bits